

# Chapter 13 Abstract Classes and Interfaces



# Motivations

You have learned how to write simple programs to create and display GUI components. Can you write the code to respond to user actions, such as clicking a button to perform an action?

In order to write such code, you have to know about interfaces. An *interface* is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.



# Objectives

To design and use abstract classes (§13.2).

To generalize numeric wrapper classes, **BigInteger**, and **BigDecimal** using the abstract **Number** class (§13.3).

To process a calendar using the **Calendar** and **GregorianCalendar** classes (§13.4).

To specify common behavior for objects using interfaces (§13.5).

To define interfaces and define classes that implement interfaces (§13.5).

To define a natural order using the **Comparable** interface (§13.6).

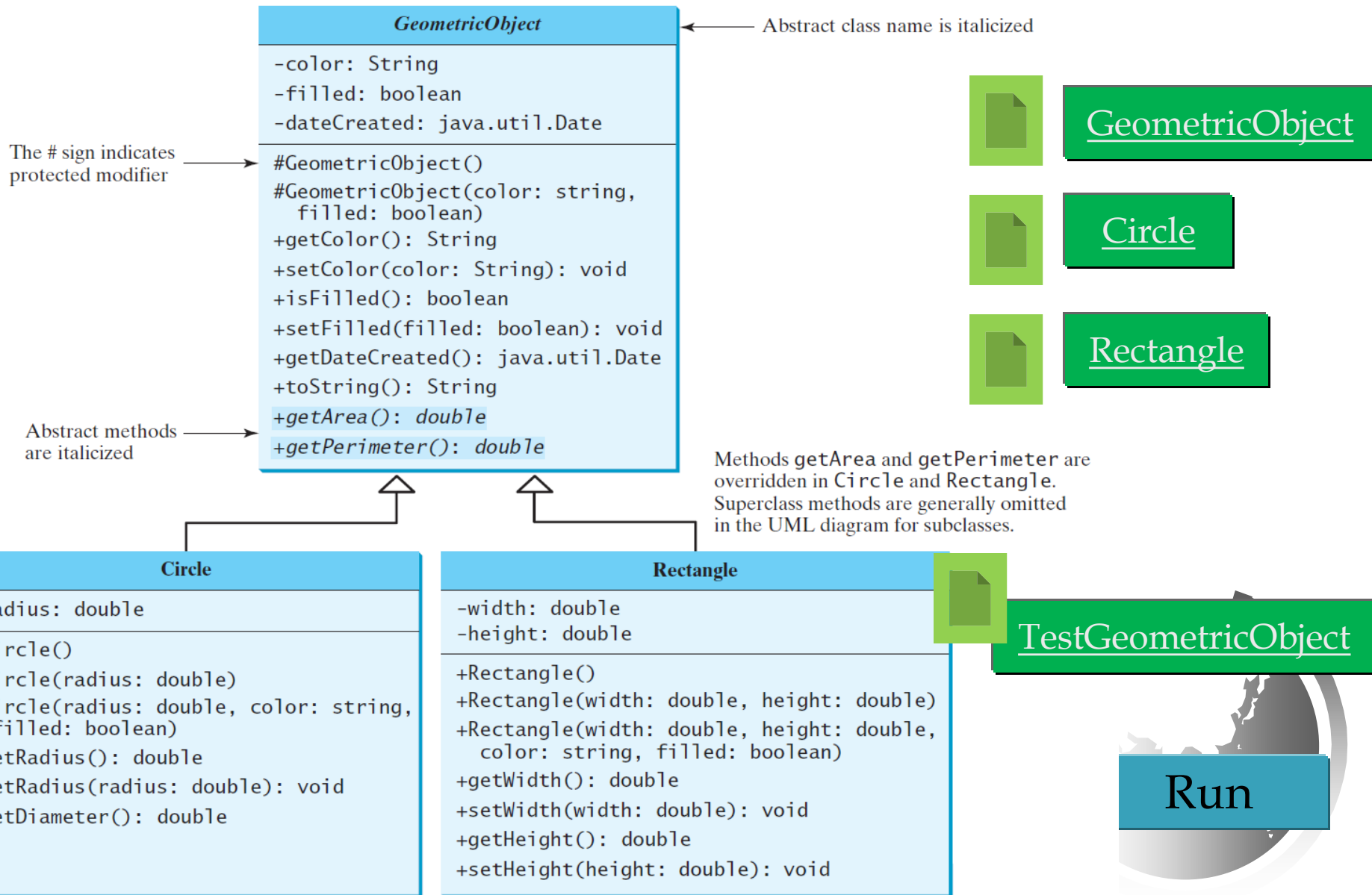
To make objects cloneable using the **Cloneable** interface (§13.7).

To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).

To design the **Rational** class for processing rational numbers (§13.9).

To design classes that follow the class-design guidelines (§13.10).

# Abstract Classes and Abstract Methods



# abstract method in abstract class

An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.



# object cannot be created from abstract class

An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.



# abstract class without abstract method

A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.



# superclass of abstract class may be concrete

A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.





# concrete method overridden to be abstract

A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.



# abstract class as type

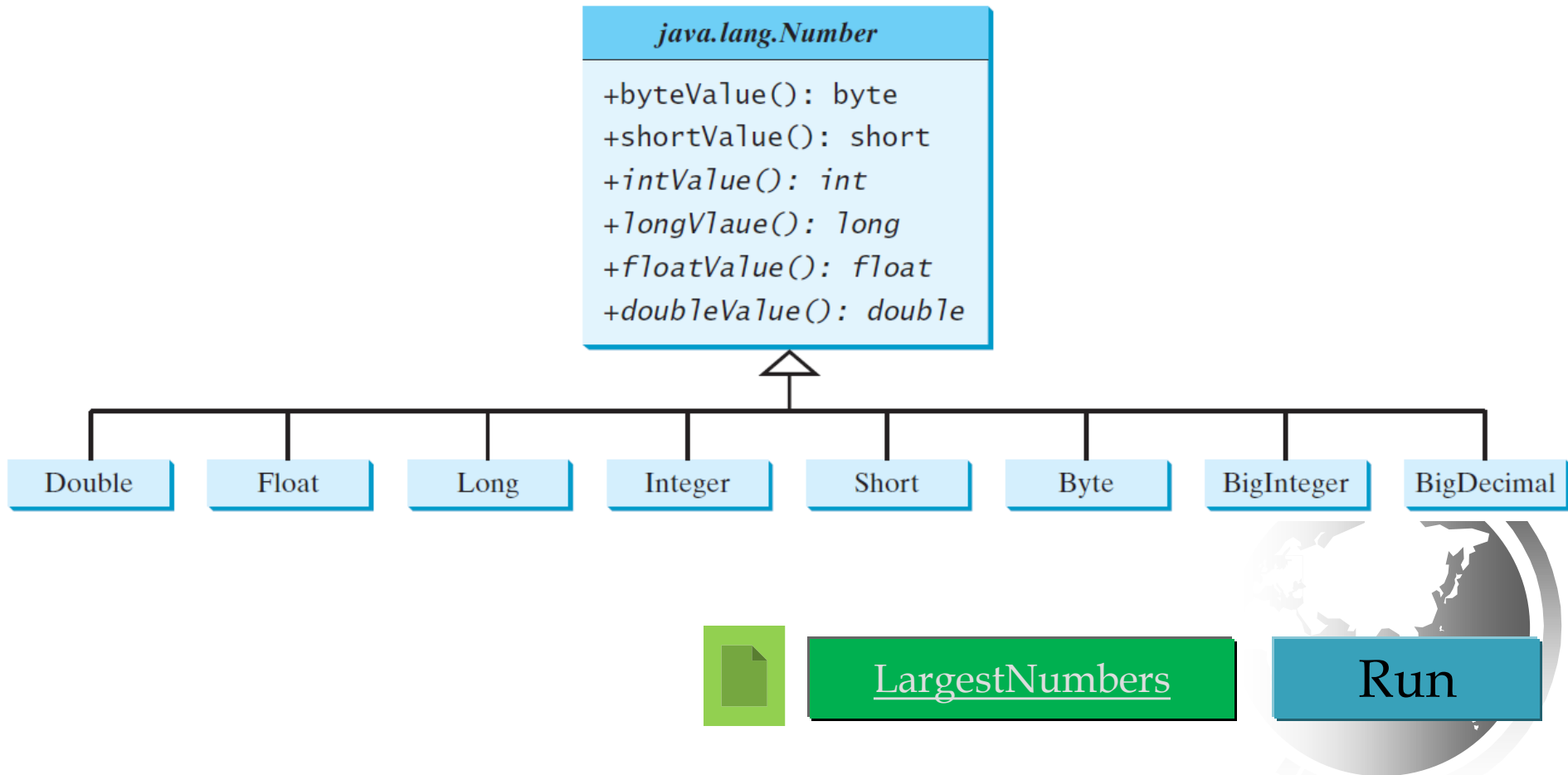
You cannot create an instance from an abstract class using the new operator, but **an abstract class can be used as a data type.**

Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new GeometricObject[10];
```



# Case Study: the Abstract Number Class



# The Abstract Calendar Class and Its GregorianCalendar subclass

## *java.util.Calendar*

```
#Calendar()  
+get(field: int): int  
+set(field: int, value: int): void  
+set(year: int, month: int,  
    dayOfMonth: int): void  
+getActualMaximum(field: int): int  
+add(field: int, amount: int): void  
+getTime(): java.util.Date  
  
+setTime(date: java.util.Date): void
```

Constructs a default calendar.

Returns the value of the given calendar field.

Sets the given calendar to the specified value.

Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.

Returns the maximum value that the specified calendar field could have.

Adds or subtracts the specified amount of time to the given calendar field.

Returns a `Date` object representing this calendar's time value (million second offset from the UNIX epoch).

Sets this calendar's time with the given `Date` object.



## *java.util.GregorianCalendar*

```
+GregorianCalendar()  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int)  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int,  
    hour: int, minute: int, second: int)
```

Constructs a `GregorianCalendar` for the current time.


Constructs a `GregorianCalendar` for the specified year, month, and date.

Constructs a `GregorianCalendar` for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

# The Abstract Calendar Class and Its GregorianCalendar subclass

An instance of `java.util.Date` represents a specific instant in time with millisecond precision.

`java.util.Calendar` is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a `Date` object. Subclasses of `Calendar` can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar. Currently, `java.util.GregorianCalendar` for the Gregorian calendar is supported in the Java API.



# The `GregorianCalendar` Class

You can use `new GregorianCalendar()` to construct a default `GregorianCalendar` with the current time and use `new GregorianCalendar(year, month, date)` to construct a `GregorianCalendar` with the specified year, month, and date. The month parameter is 0-based, i.e., 0 is for January.



# The get Method in Calendar Class

The `get(int field)` method defined in the `Calendar` class is useful to extract the date and time information from a `Calendar` object. The fields are defined as constants, as shown in the following.

| <i>Constant</i>      | <i>Description</i>  |
|----------------------|---|
| <b>YEAR</b>          | The year of the calendar.   |
| <b>MONTH</b>         | The month of the calendar, with 0 for January.                    |
| <b>DATE</b>          | The day of the calendar.  |
| <b>HOURL</b>         | The hour of the calendar (12-hour notation).                      |
| <b>HOURL_OF_DAY</b>  | The hour of the calendar (24-hour notation).                      |
| <b>MINUTE</b>        | The minute of the calendar.                                       |
| <b>SECOND</b>        | The second of the calendar.                                       |
| <b>DAY_OF_WEEK</b>   | The day number within the week, with 1 for Sunday.                |
| <b>DAY_OF_MONTH</b>  | Same as <b>DATE</b> .   |
| <b>DAY_OF_YEAR</b>   | The day number in the year, with 1 for the first day of the year. |
| <b>WEEK_OF_MONTH</b> | The week number within the month, with 1 for the first week.      |
| <b>WEEK_OF_YEAR</b>  | The week number within the year, with 1 for the first week.       |
| <b>AM_PM</b>         | Indicator for AM or PM (0 for AM and 1 for PM).                   |



# Getting Date/Time Information from Calendar



TestCalendar

Run



# Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?



# What is an interface?

## Why is an interface useful?

An interface is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.



# Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {  
    constant declarations;  
    abstract method signatures;  
}
```

Example:

```
public interface Edible {  
    /** Describe how to eat */  
    public abstract String howToEat();  
}
```



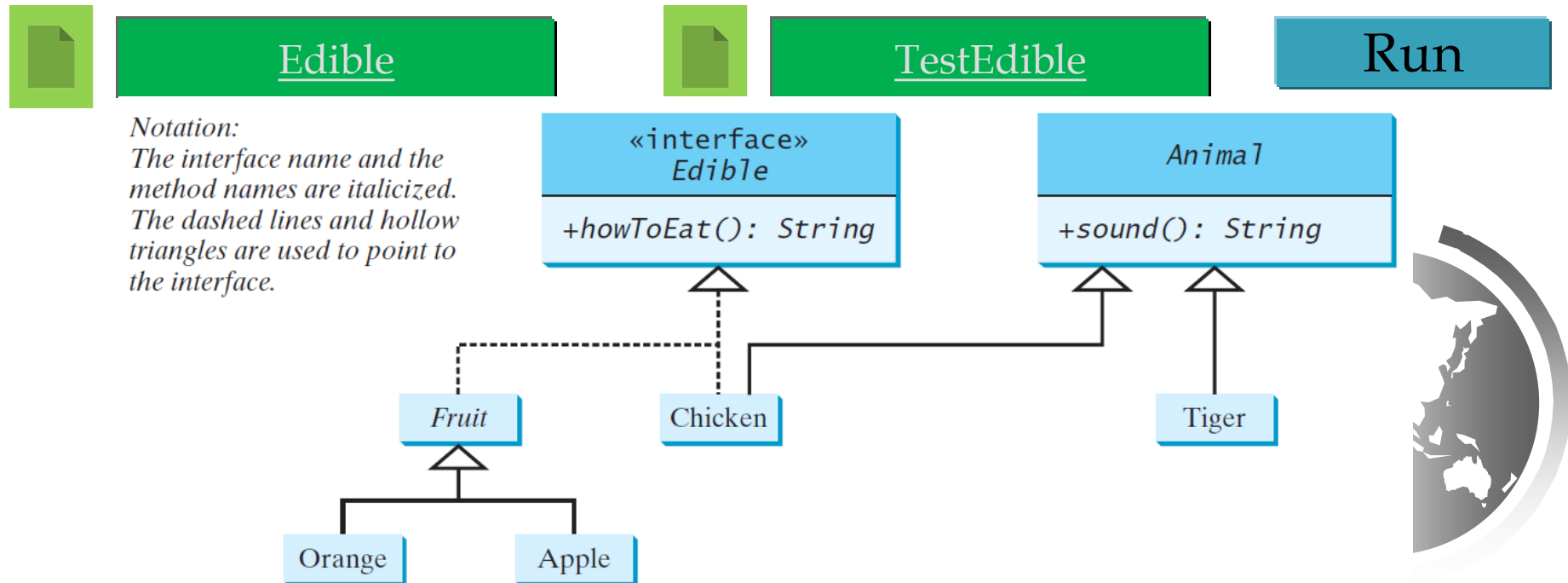
# Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable, as the result of casting, and so on.



# Example

You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).



# Omitting Modifiers in Interfaces

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {  
    public static final int K = 1;  
  
    public abstract void p();  
}
```

Equivalent

```
public interface T1 {  
    int K = 1;  
  
    void p();  
}
```

A constant defined in an interface can be accessed using syntax `InterfaceName.CONSTANT_NAME` (e.g., `T1.K`).



# Example: The Comparable Interface

```
// This interface is defined in  
// java.lang package  
package java.lang;
```

```
public interface Comparable<E> {  
    public int compareTo(E o);  
}
```



# The toString, equals, and hashCode Methods

Each wrapper class overrides the `toString`, `equals`, and `hashCode` methods defined in the `Object` class. Since all the numeric wrapper classes and the `Character` class implement the `Comparable` interface, the `compareTo` method is implemented in these classes.





# Integer and BigInteger Classes

```
public class Integer extends Number
    implements Comparable<Integer> {
    // class body omitted

    @Override
    public int compareTo(Integer o) {
        // Implementation omitted
    }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
    // class body omitted

    @Override
    public int compareTo(BigInteger o) {
        // Implementation omitted
    }
}
```

# String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
    // class body omitted

    @Override
    public int compareTo(String o) {
        // Implementation omitted
    }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
    // class body omitted

    @Override
    public int compareTo(Date o) {
        // Implementation omitted
    }
}
```

# Example

```
1 System.out.println(new Integer(3).compareTo(new Integer(5)));  
2 System.out.println("ABC".compareTo("ABE"));  
3 java.util.Date date1 = new java.util.Date(2013, 1, 1);  
4 java.util.Date date2 = new java.util.Date(2012, 1, 1);  
5 System.out.println(date1.compareTo(date2));
```



# Generic sort Method

Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object. All the following expressions are **true**.

```
n instanceof Integer  
n instanceof Object  
n instanceof Comparable
```

```
s instanceof String  
s instanceof Object  
s instanceof Comparable
```

```
d instanceof java.util.Date  
d instanceof Object  
d instanceof Comparable
```

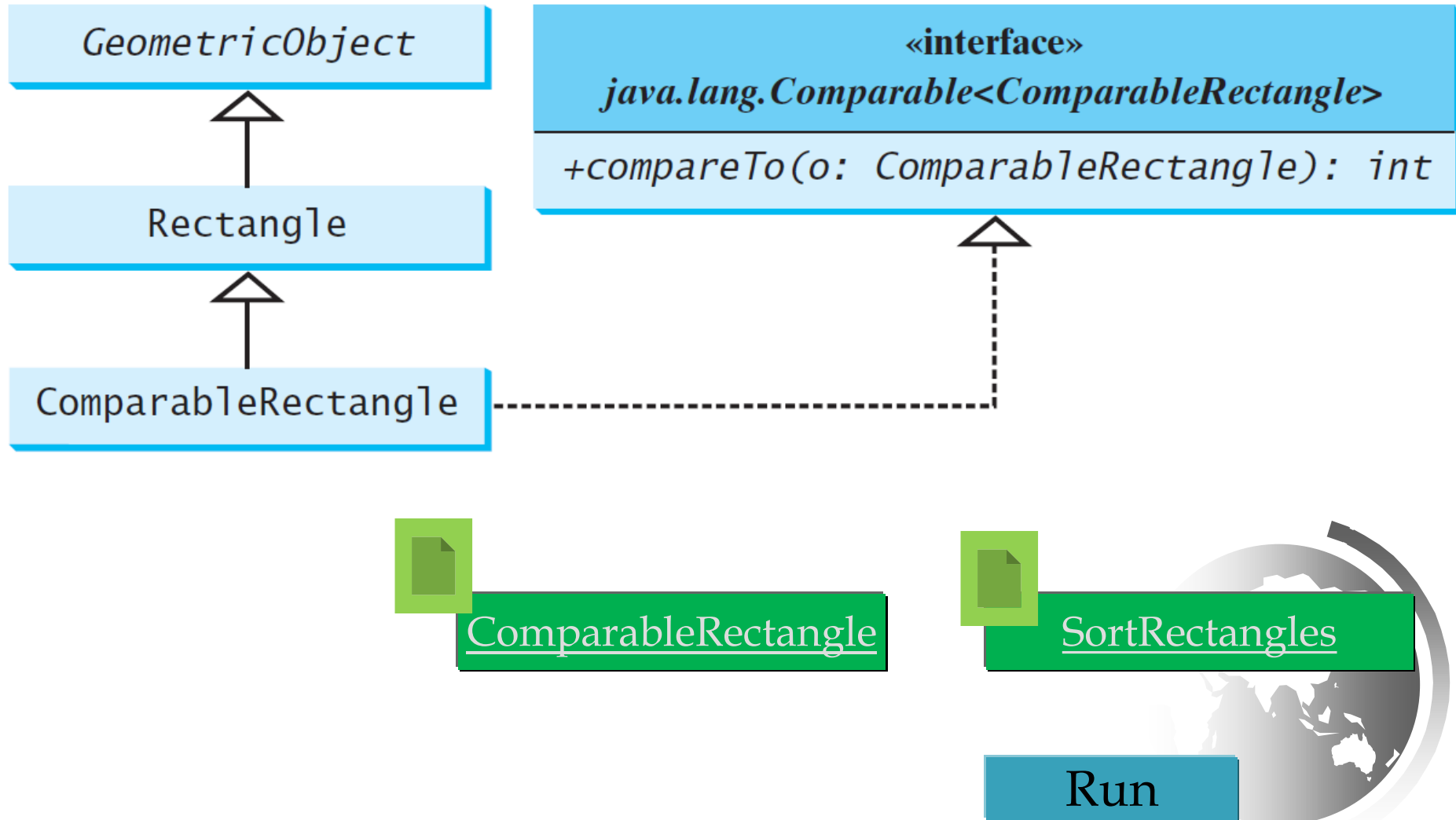
The `java.util.Arrays.sort(array)` method requires that the elements in an array are instances of `Comparable<E>`.



SortComparableObjects

Run

# Defining Classes to Implement Comparable



# The Cloneable Interfaces

Marker Interface: An empty interface.

A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

```
package java.lang;  
public interface Cloneable {  
}
```



# Examples

Many classes (e.g., Date and Calendar) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);  
Calendar calendarCopy = (Calendar)calendar.clone();  
System.out.println("calendar == calendarCopy is " +  
    (calendar == calendarCopy));  
System.out.println("calendar.equals(calendarCopy) is " +  
    calendar.equals(calendarCopy));
```

displays

calendar == calendarCopy is false

calendar.equals(calendarCopy) is true



# Implementing Cloneable Interface

To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class. The following code defines a class named House that implements Cloneable and Comparable.

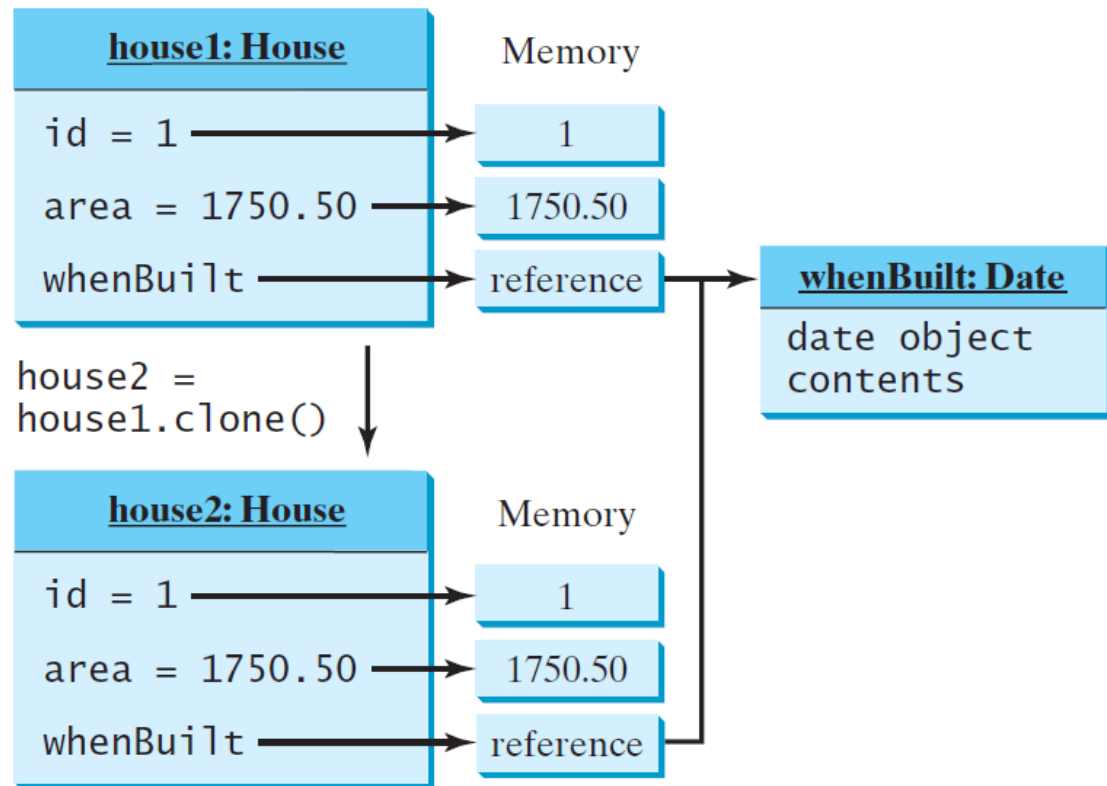


# Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);
```

```
House house2 = (House)house1.clone();
```

## Shallow Copy



(a)

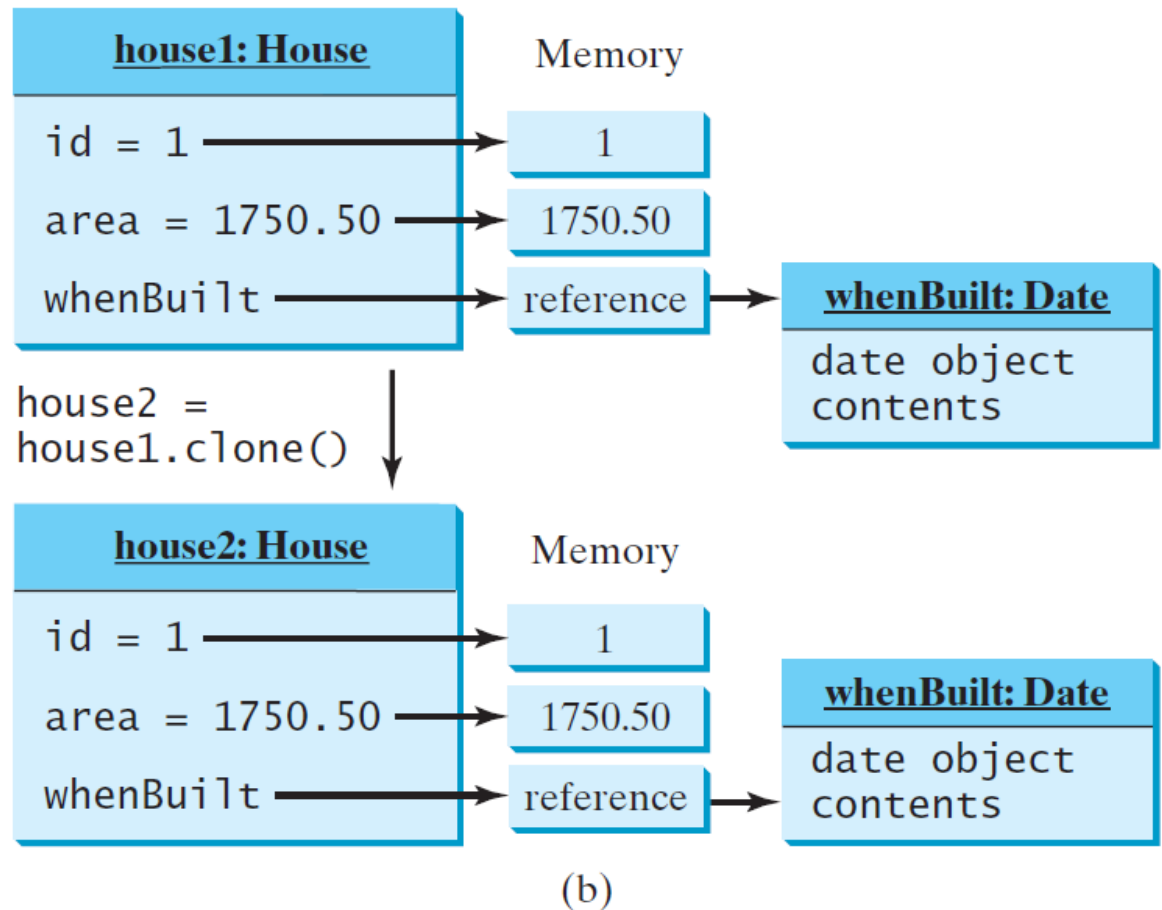


# Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);
```

```
House house2 = (House)house1.clone();
```

Deep  
Copy



# Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

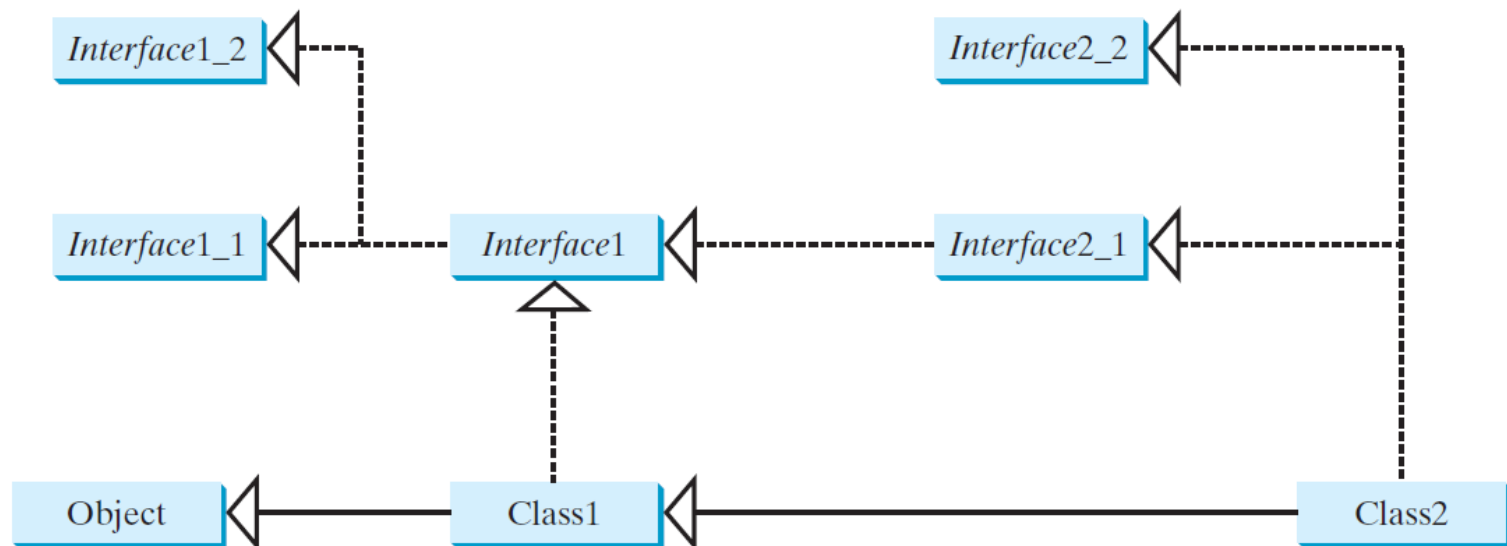
Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

|                | <i>Variables</i>                                   | <i>Constructors</i>   | <i>Methods</i>                                       |
|----------------|--|---|--|
| Abstract class | No restrictions.                                   | Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator. | No restrictions.                                     |
| Interface      | All variables must be <b>public static final</b> . | No constructors. An interface cannot be instantiated using the new operator.  | All methods must be public abstract instance methods |



# Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.



Suppose that *c* is an instance of *Class2*. *c* is also an instance of *Object*, *Class1*, *Interface1*, *Interface1\_1*, *Interface1\_2*, *Interface2\_1*, and *Interface2\_2*.

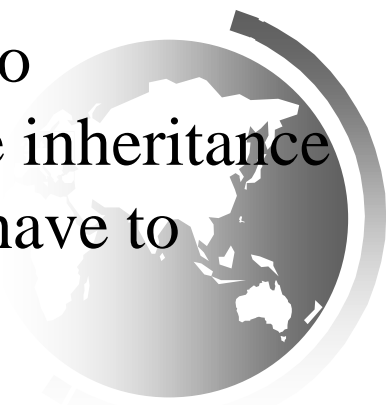
# Caution: conflict interfaces

In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type). This type of errors will be detected by the compiler.



# Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface.



# The Rational Class

`java.lang.Number`

`Rational`

`java.lang.Comparable<Rational>`

1

1

Add, Subtract, Multiply, Divide

## Rational

-numerator: long  
-denominator: long

+Rational()

+Rational(numerator: long,  
denominator: long)

+getNumerator(): long

+getDenominator(): long

+add(secondRational: Rational):  
Rational

+subtract(secondRational:  
Rational): Rational

+multiply(secondRational:  
Rational): Rational

+divide(secondRational:  
Rational): Rational

+toString(): String

-gcd(n: long, d: long): long

The numerator of this rational number.

The denominator of this rational number.

Creates a rational number with numerator 0 and denominator 1.

Creates a rational number with a specified numerator and denominator.

Returns the numerator of this rational number.

Returns the denominator of this rational number.

Returns the addition of this rational number with another.

Returns the subtraction of this rational number with another.

Returns the multiplication of this rational number with another.

Returns the division of this rational number with another.

Returns a string in the form "numerator/denominator." Returns the numerator if denominator is 1.

Returns the greatest common divisor of n and d.

Rational

TestRationalClass

Run

# Designing a Class

(Coherence) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.



# Designing a Class, cont.

(Separating responsibilities) A single entity with too many responsibilities can be broken into several classes to separate responsibilities. The classes `String`, `StringBuilder`, and `StringBuffer` all deal with strings, for example, but have different responsibilities. The `String` class deals with immutable strings, the `StringBuilder` class is for creating mutable strings, and the `StringBuffer` class is similar to `StringBuilder` except that `StringBuffer` contains synchronized methods for updating strings.





# Designing a Class, cont.

Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.



# Designing a Class, cont.

Provide a public no-arg constructor and override the equals method and the toString method defined in the Object class whenever possible.



# Designing a Class, cont.

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. Always place the data declaration before the constructor, and place constructors before methods. Always provide a constructor and initialize variables to avoid programming errors.



# Using Visibility Modifiers

Each class can present two contracts – one for the users of the class and one for the extenders of the class. Make the fields private and accessor methods public if they are intended for the users of the class. Make the fields or method protected if they are intended for extenders of the class. The contract for the extenders encompasses the contract for the users. The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but you should never change the implementation in a way that violates that contract.



# Using Visibility Modifiers, cont.

A class should use the private modifier to hide its data from direct access by clients. You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify. A class should also hide methods not intended for client use. The gcd method in the Rational class is private, for example, because it is only for internal use within the class.



# Using the static Modifier

A property that is shared by all the instances of the class should be declared as a static property.

